Rapport de projet de Logique et Modèles de Calcul

Algorithme d'unification Martelli-Montanari

M1 Informatique FST NANCY
M. GALMICHE

BOUR Camille
SOMMERLATT Justine

Table des matières

I. Introduction

Dans le cadre de notre première année de Master en Informatique, il nous est demandé d'effectuer une implantation de l'algorithme d'unification de Martelli-Montanari pour l'Unité d'Enseignement Logique et Modèles de Calcul enseignée par M. GALMICHE Didier.

L'algorithme d'unification de Martelli-Montanari est un algorithme sous forme de différentes règles, visant à transformer un ensemble d'équations données.

Les règles sont :

supprimer	$G \cup \{ t \doteq t \} \Rightarrow G$	
décomposer	$G \cup \{ f(s_1,...,s_k) \doteq f(t_1,...,t_k) \} \Rightarrow G \cup \{ s_1 \doteq t_1, ..., s_k \doteq t_k \}$	
échanger	$G \cup \{ f(\vec{t}) \doteq x \} \Rightarrow G \cup \{ x \doteq f(\vec{t}) \}$	
éliminer	$G \cup \{ x \doteq t \} \Rightarrow G\{x \mapsto t\} \cup \{ x \doteq t \}$	si $x \notin vars(t)$ et $x \in vars(G)$

Ce rapport visant à justifier nos choix d'implantation et à décrire les règles utilisées, s'inscrit donc dans la résolution de ce projet.

II. Les règles de transformation

Dans la suite de ce rapport, on définit une équation E telle que E est du type X?=T.

1. Les prédicats pour les règles

a) Renommage de variables

La règle de renommage de variable permet d'évaluer si « rename » peut être appliqué sur une équation donnée, s'appliquant uniquement à deux variables.
Cette règle est définie comme suit :

```
regles(X?=T,rename) :- var(X),var(T),!.
```

Pour une équation E définie comme X?=T on vérifie que les deux arguments X et T de notre équation correspondent bien à des variables à l'aide de var(X) et var(T), s'il s'agit effectivement de variables, on stoppe l'exécution de l'arbre.

b) Règle d'échange

La règle d'échange permet d'évaluer si « orient » peut être appliqué sur une équation donnée, s'appliquant sur un terme instancié et une variable.
Cette règle est définie comme suit :

```
regles(T?=X,orient) :- var(X),nonvar(T),!.
```

Pour notre équation T?=X définie ci-dessus, on vérifie que l'argument T est un terme instancié et l'argument X une variable, si T est bien un terme instancié et X une variable alors on stoppe l'exécution de l'arbre.

c) Règle de simplification

La règle de simplification permet de définir si « simplify » peut être appliqué sur une équation donnée, elle s'applique dans deux cas : sur une variable et une constante ou sur deux constantes.

```
regles(X?=A,simplify) :- var(X),atomic(A),!;atomic(X),atomic(A),X==A,!.
```

Pour notre équation X?=A, on vérifie tout d'abord que X est une variable et A une constante, si c'est le cas alors on stoppe l'exécution de l'arbre. Sinon on vérifie que X et A sont tous les deux des constantes, si oui on stoppe l'exécution de l'arbre. L'exécution de l'arbre s'arrête lorsque l'une de ses deux clauses est vérifiée.

d) Règle de développement

La règle de développement permet de définir si « expand » peut être appliqué à une équation donnée, elle s'applique sur une variable et un terme composé.

```
regles(X?=T,expand) :- var(X),compound(T),\+occur_check(X,T),!.
```

Pour exemple X ?=f(G), X ?=f(h(F))
On vérifie tout d'abord que X est une variable avec var(X) et que T est bien un terme composé grâce à compound(T). Il faut également vérifier que la variable X n'apparaît pas dans le terme composé T, pour cela on utilise occur_check qui renvoie vrai si X apparaît dans T. On ajoute donc la négation devant occur_check afin d'avoir true lorsque X n'apparaît pas dans T (\+occur_check(X,T)), si toutes les clauses sont vraies, on stoppe l'exécution de l'arbre.

e) Règle de décomposition

La règle de décomposition permet de définir si « decompose » peut être appliqué à une équation donnée, elle s'applique uniquement sur deux termes composés.

```
regles(X?=T,decompose) :-
compound(X),compound(T),functor(X,N,A),functor(T,M,B),N==M,A==B,!.
```

Pour exemple f(X)?=f(T), f(f(X),G)?=f(G,D)
On vérifie que les termes X et T sont bien des termes composés à l'aide de compound(X) et compound(T). On appelle ensuite le prédicat functor qui renseigne le nom de la fonction ainsi que le nombre d'arguments sur laquelle il est appelé, on récupère donc les noms des fonctions ainsi que leur nombre d'arguments à l'aide de functor(X,N,A) et functor(T,M,B) afin de les comparer. Si N et M sont égaux et que A et B sont égaux cela signifie que les noms ainsi que le nombre d'arguments des fonctions sont identiques, on stoppe donc l'exécution de l'arbre.

f) Règle de gestion de conflit

La règle de gestion de conflit (ou règle de collision) permet de définir si « clash » peut être appliqué à une équation donnée, elle s'applique uniquement sur deux termes composés.

```
regles(X?=T,clash) :- compound(X),compound(T),functor(X,N,A),functor(T,M,B),
(N \== M ; A \== B),!.
```

Pour exemple f(X)?=g(T), f(f(X),G)?=f(G)
On vérifie tout d'abord que les termes X et T sont bien des termes composés à l'aide de compound(X) et compound(T).
On appelle le prédicat functor qui renseigne le nom de la fonction ainsi que le nombre d'arguments sur laquelle il est appelé, on récupère donc les noms des fonctions ainsi que leur nombre d'arguments à l'aide de functor(X,N,A) et functor(T,M,B) afin de les comparer. Pour que cette règle renvoie true, il faut que les noms des fonctions soient différents ou que les

nombres d'arguments des fonctions soient différents, c'est pourquoi on utilise (N \== M ; A \== B), qui correspond à la négation de l'égalité des fonctions, soit leur inégalité.

g) Règle d'occurrence

La règle de vérification permet de définir si « check » peut être appliqué sur une équation donnée, elle s'applique sur une variable et une variable ou terme quelconque.

```
regles(X?=T,check) :- var(X),X\==T,!,occur_check(X,T).
```

On vérifie que X soit bien une variable à l'aide de var(X). On compare X et T afin de vérifier qu'ils soient bien différents (pour ne pas rechercher une occurrence de X dans X). Si une occurrence de X est trouvée dans l'équation, on stoppe l'exécution de l'arbre.

2. Le prédicat d'occurrence occur_check

Le prédicat d'occurrence cherche à déterminer si une variable V apparaît dans un terme composé T. Il prend donc en paramètre une variable et un terme composé, et renvoie vrai si cette variable apparaît dans le terme composé.

```
occur_check(V,T) :-   var(T),T==V,!.
occur_check(V,T) :-   compound(T),arg(_,T,X),occur_check(V,X),!.
```

Tout d'abord, si T est une variable, on teste l'égalité avec V, si ils sont égaux alors on stoppe l'exécution de l'arbre. Sinon, si T est un terme composé on cherche à explorer tous ses arguments afin de définir s'il y a présence d'occurrence. Pour cela, on appelle arg(_,T,X) : le symbole _ est une variable anonyme qui sert à ignorer la valeur, arg(_T,X) va donc générer des branches pour tous les arguments de T, tant que le test occur_check(V,X) n'est pas vérifié, on recherche l'occurrence sur les branches une par une, il s'agit d'un appel récursif.

3. Le prédicat de réduction reduit

Le prédicat de réduction prend quatre paramètres :

- une règle
- une équation
- un système d'équation
- une variable (qui sera le résultat de l'application de la règle sur l'équation)

```
reduit(R,E,P,Q) :-  \+regles(E,clash),\+regles(E,check),regles(E,R),aff_regle(R,E),!,
                                application(R,E,P,Q).
```

Tout d'abord, on doit vérifier que les règles d'occurrence et de conflit ne s'appliquent pas. Pour cela on appelle la négation des règles clash et check, soit \+regles(E,clash) et \+regles(E,check). Lorsque l'on trouve une règle à appliquer pour l'équation E (regles(E,R)), on active la coupure tout en appliquant la règle sur le système d'équation (application(R,E,P,Q)).

4. Le prédicat d'application des règles

On souhaite appliquer une règle R à une équation E grâce à ce prédicat.
Le prédicat d'application des règles :

- une règle R
- une équation E
- un système d'équation P
- une variable (qui sera le résultat de l'application de la règle sur l'équation) Q

application(R,E,P,Q) applique la règle R à l'équation E (qui est comprise dans le système d'équations P), et place la réponse dans la variable Q.
La règle R n'est appliquée uniquement si les conditions sont réunies pour l'appliquer, les vérifications étant déjà faites, il n'est pas nécessaire d'en refaire dans ces prédicats.

a) Renommage de variables

R : rename
E : X?=T

```
application(rename,X?=T,P,Q) :- X=T,Q=P..
```

Les tests de validité étant déjà faits, X est directement remplacé par T dans toutes les équations de P, et on affecte P à Q.

b) Règle d'échange

R : orient
E : X?=T

```
application(orient,T?=X,P,Q) :- append([X?=T],P,Q).
```

Les tests de validité étant déjà faits et l'équation E étant de type X?=T, on concatène T ?=X à P et on stocke le résultat dans Q.

c) Règle de simplification

R : simplify
E : X?=T

```
application(simplify,X?=T,P,Q) :- X=T,Q=P.
```

Les tests de validité étant déjà faits, X est remplacé par T dans toutes les équations de P, et on affecte P à Q.

d) Règle de développement

R : expand
E : X?=T

```
application(expand,X?=T,P,Q) :- X=T,Q=P.
```

Les tests de validité étant déjà faits, X est remplacé par T dans toutes les équations de P, et on affecte P à Q.

e) Règle de décomposition

R : decompose
E : X?=T

```
application(decompose,X?=T,P,Q) :-
X=..XT,new_list(XT,XL),T=..TT,new_list(TT,TL),croisement(XL,TL,S),append(S,P,Q).
```

Les arguments sont des termes de type f(A,B,C) avec A, B et C pouvant être des variables ou des termes quelconques.
Une liste T récupère le nom et les arguments de la fonction, et une autre liste L récupère le uniquement les arguments de la fonction car le nom est déjà recueilli dans T et n'a pas de rôle ici. On fait de même pour le deuxième argument de l'équation (new_list(TT,TL)). On croise les deux listes obtenues à partir des deux arguments de l'équation dans l'objectif d'obtenir une nouvelle liste S de type [A ?=B,C ?=D,…] que l'on concatène à la liste d'équations P et on stocke le résultat dans Q.

f) Règle de gestion de conflit

R : clash

```
application(clash,_,_,_) :-  fail.
```

Lorsque cette règle peut s'appliquer cela signifie que le système d'équation est incorrect. L'appel de fail fait échouer le programme.
Définir cette règle n'est pas obligatoire, puisque l'absence de clause signifie un fail.

g) Règle d'occurrence

R : check

```
application(check,_,_,_) :-  fail.
```

Lorsque cette règle peut s'appliquer cela signifie que le système d'équation est incorrect. L'appel de fail fait échouer le programme.
Définir cette règle n'est pas obligatoire, puisque l'absence de clause signifie un fail.

5. Le prédicat d'unification

Le prédicat d'unification unifie(P) permet d'unifier le système d'équations P.

```
unifie([A|P]) :-  aff_sys([A|P]), reduit(_,A,P,Q),!, unifie(Q).
unifie([]) :-  echo('\n Il n'y a plus d'équation à unifier, unification terminée\n').
```

On appelle reduit(_,A,P,Q) qui applique une règle sur l'équation A du système d'équations P et stocke le résultat dans Q, puis on appelle récursivement unifie sur Q. L'objectif est d'appeler unifie récursivement jusqu'à ce que Q soit vide car lorsque l'on appelle unifie sur une liste vite cela signifie qu'il n'y a plus d'équations à unifier et donc que le programme est terminé, ce qui va laisser place à l'affichage (echo).

III. Prédicats et expressions prédéfinis utilisés

- **Le prédicat arg(…)**

Le prédicat arg(..) permet de récupérer l'argument d'une équation.
Le prédicat arg(n,E,X) récupère le n-ième argument de l'équation E et de le stocker dans X. (avec n un entier strictement supérieur à 0)

- **Le prédicat functor(…)**

Le prédicat functor(…) permet de récupérer le nom d'une fonction ainsi que son nombre d'arguments.
Le prédicat functor(X,A,N) récupère le nom de la fonction X et de le stocker dans A, ainsi que son nombre d'argument qu'il stocke dans N.

- **Le prédicat var(…)**

Le prédicat var(…) permet de définir si un terme est une variable.
Le prédicat var(X) renvoie true si X est une variable, sinon renvoie false.

- **Le prédicat nonvar(…)**

Le prédicat nonvar(…) permet de définir si un terme est un terme instancié.
Le prédicat nonvar(X) renvoie true si X est un terme instancié (n'est pas une variable), sinon renvoie false.

- **Le prédicat atomic(…)**

Le prédicat atomic(…) permet de définir si un terme est une constante.
Le prédicat atomic(X) renvoie true si X est une constante, sinon renvoie false.

- **Le prédicat compound(…)**

Le prédicat compound permet de définir si un terme est composé.
Le prédicat compound(X) renvoie true si X est un terme composé, sinon renvoie false.

- **Le prédicat =**

Le prédicat = permet d'unifier un terme.
X=T signifie donc que X s'unifie à T et que tous les X sont remplacés par des T.

- **L'expression X=..XT**

L'expression X=..XT permet de décomposer un terme en une liste.
X=..XT décompose le terme X en une liste XT.
Par exemple : X = f(B,J,R)
X=..XT
XT = [f,B,J,R]

- **Le prédicat new_list(…)**

Le prédicat new_list(…) permet de créer depuis une liste existante, une seconde liste sans le premier élément de l'ancienne liste.
Ce prédicat permet de contrer le fait que l'appel de X=..XT inclut également le nom de la fonction (si le nom de la fonction n'est pas désiré).

- **Le prédicat croisement(…)**

Le prédicat croisement(…) permet de « croiser » deux listes en une seule liste.
Ce prédicat sélectionne deux à deux les éléments de deux listes et les transforme en un nouvel élément de la forme X?=T (avec X l'élément de la première liste et T l'élément de la deuxième liste) ajouté à une liste résultat. On réitère l'opération jusqu'à ce que les listes soient vides.

- **La coupure !**

La coupure ! permet de ne pas continuer l'exploration d'autre branches de l'arbres si la branche courante renvoie true.

IV. Affichage et initialisation du programme

1) Affichage du programme

Ici S est le choix d'unification de l'utilisateur.

- **Activation de l'affichage**

Le code qui permet l'affichage est : trace_unif(P,S) :- set_echo,unifie(P,S).

- **Suppression de l'affichage**

Le code qui inhibe l'affichage est : unif_(P,S) :- clr_echo,unifie(P,S).

2) Initialisation du programme

Dès le lancement de swi-prolog, le main est lancé grâce à la commande :

```
:- initialization main.
```

Lorsque le main est lancé, cela provoque un affichage censé guidé l'utilisateur sur la manière d'utiliser le programme.

```
main :- write('\n\n\n\nAlgorithme d\'unification de Martelli_Montanari\n\n'),
        write('L\'affichage des traces d\'exécution est activé par défaut si vous saisissez
unifie(P,S)\n\n'),
        write('Pour désactiver l\'affichage et exécuter sans aucune trace d\'unification, saisir
unif(P,S).\n\n'),
        write('Pour réactiver l\'affichage et exécuter avec traces d\'exécution, saisir
trace_unif(P,S).\n\n'),
        write('P est un système d\'équation de type [X?=Y,f(G)?=Z]\n'),
        write('S est la stratégie souhaitée : choix_premier ou choix_pondere\n\n\n\n'),
        set_echo.
```

L'utilisateur peut choisir d'exécuter le programme avec ou sans traces d'exécution.

3) Traces d'exécution du programme

Afin de fournir des traces d'exécution du programme, le choix s'est porté sur un exemple qui se voulait significatif :

?- unifie ([f(X,Y) ?= f(Z,J) , Z ?= f(Y) , M ?= t] , CHOIX).

- **Exécution avec choix premier**

choix_premier([E|P],Q,E,R) :- reduit(R,E,P,Q),!.

Choisit la première équation du système afin de la réduire.

```
?- trace_unif( [ f(X,Y) ?= f(Z,g) , Z ?= f(Y) , M ?= t ] ,
choix_premier ).
system: [f(_12860,_12862)?=f(_12866,g),_12866?=f(_12862),_12900?=t]
decompose: f(_12860,_12862)?=f(_12866,g)
system: [_12860?=_12866,_12862?=g,_12866?=f(_12862),_12900?=t]
rename: _12860?=_12866
system: [_12862?=g,_12860?=f(_12862),_12900?=t]
simplify: _12862?=g
system: [_12860?=f(g),_12900?=t]
expand: _12860?=f(g)
system: [_12900?=t]
simplify: _12900?=t

 Unification teminée.
Résultat :
X = Z, Z = f(g),
Y = g,
M = t.
```

- **Exécution avec choix pondéré**

choix_pondere(P,Q,E,R) :- choix_eq(P,Q,E,R,[]),!.

Choisit la règle applicable avec le poids le plus élevé afin de l'appliquer en premier, pour cela il nous faut faire appel à choix_eq qui compare deux à deux des équations afin de définir l'équation comportant les règles à appliquer en priorité. choix_eq fait elle-même appel à choix_regle qui décrit l'ordre de priorité des règles selon l'ordre de grandeur donné dans le sujet :

clash; check > rename; simplify > orient > decompose > expand

```
?- trace_unif( [ f(X,Y) ?= f(Z,g) , Z ?= f(Y) , M ?= t ] ,
choix_pondere ).
system: [f(_8468,_8470)?=f(_8474,g),_8474?=f(_8470),_8508?=t]
simplify: _8508?=t
system: [f(_8468,_8470)?=f(_8474,g),_8474?=f(_8470)]
decompose: f(_8468,_8470)?=f(_8474,g)
system: [_8468?=_8474,_8470?=g,_8474?=f(_8470)]
rename: _8468?=_8474
system: [_8468?=f(_8470),_8470?=g]
simplify: _8470?=g
system: [_8468?=f(g)]
expand: _8468?=f(g)

 Unification teminée.
Résultat :
X = Z, Z = f(g),
Y = g,
M = t.
```

- **Choix aléatoire**

Comme autre possibilité d'exécution, le choix aléatoire aurait pu être une solution envisageable pour l'unification.

choix_alea(P,Q,E,R) :-length(P,A), C is random(A)+1,arg(C,P,G),reduit(R,G,P,Q),!.

V. Sources utilisées pour le projet

Tutoriel SWI-prolog
https://www.youtube.com/watch?v=t6L7O7KiE-Q

Tutoriel SWISH—SWI-Prolog
https://www.youtube.com/watch?v=g34jS24JWqw

Un cours sur les premiers pas en Prolog de l'université de Lyon
https://perso.liris.cnrs.fr/nathalie.guin/Prolog/Cours/Cours2-Prolog1.pdf

Un forum contenant une explication pour la différence entre = et == en prolog
https://stackoverflow.com/questions/8219555/what-is-the-difference-between-and-in-prolog/8220315

Un site internet regroupant plusieurs cours et articles d'aide pour prolog
https://prolog.developpez.com/cours/#deb

WIKIPEDIA